Hardware design to accelerate PNG encoder for binary mask compression on FPGA

Rostom Kachouri, Mohamed Akil

Hardware design to accelerate PNG encoder for binary mask compression on FPGA

Rostom Kachouri
ESIEE Paris, LIGM, A3SI,
2 Bd Blaise Pascal, BP 99,
93162 Noisy-Le-Grand, France
Email: rostom.kachouri@esiee.fr

Mohamed Akil
ESIEE Paris, LIGM, A3SI,
2 Bd Blaise Pascal, BP 99,
93162 Noisy-Le-Grand, France
Email: mohamed.akil@esiee.fr

***Abstract*—PNG (Portable Network Graphics) is a lossless compression method for real-world pictures. Since its specification, it continues to attract the interest of the image processing community. Indeed, PNG is an extensible file format for portable and well-compressed storage of raster images. In addition, it supports all of Black and White (binary mask), grayscale, indexed-color, and truecolor images. Within the framework of the $Demat+$ project which intend to propose a complete solution for storage and retrieval of scanned documents, we address in this paper a hardware design to accelerate the PNG encoder for binary mask compression on FPGA. For this, an optimized architecture is proposed as part of an hybrid software and hardware co-operating system. For its evaluation, the new designed PNG IP has been implemented on the *ALTERA "Arria II GX EP2AGX125EF35"* FPGA. The experimental results show a good match between the achieved compression ratio, the computational cost and the used hardware resources.**

***Keywords*—PNG encoder, Prediction, LZ77, Huffman, binary masks, Hardware design, FPGA.**

I. Introduction

Within the framework of the *Demat+*[1] project, we aim to propose a complete solution for storage and retrieval of scanned documents. In this context, the intended paperless application to achieve have to transmit the scanned documents with a low-bandwidth network to a computer cloud. The overall idea consist to partition the source document into three layers: a foreground layer, a background layer, and a binary mask [7], and then to use different compression strategies for the same document. In literature, the often used taxonomy distinguishes two categories of image compression format: the lossless compression formats and the lossy ones [2]. The lossless compression formats perform compression on the image matrix. It is worth noting that the transformation between a raw format and the lossless compression one is bijective [5], [3]. Which means that when decompressing a lossless compressed image, the original image is restored, and it is a 100% identical copy of the original. On the other side, the lossy compression formats achieve better compression rate at the cost of image degradation [4], [6]. A quantification stage is applied in this case on the frequency transform of the image[2]. We note that the foreground and background layers which contain, respectively, the color information of the text and the original background of the image can be compressed via a lossy compression format like JPEG [4] for example. By against, the binary mask, where the text, and possibly pieces of thin strokes when they exist, are located is necessarily subjected to a non-destructive compression. One of the most interesting lossless compression formats is PNG (Portable Network Graphics). In fact, it provides a portable, legally unencumbered, well-compressed, well-specified standard for lossless bitmapped image files [5]. To respond to real-time constraints, aimed to be respected by the *SagemCom* company, we expect in this paper to accelerate the employed PNG encoder for binary mask compression through a hardware implementation. Indeed, while compare with software encoding, parallel processing is the most significant feature of high efficiency. Front of a custom development, based on ASIC technology, the capacity and performance of current FPGAs are such that they present a much more realistic alternative than they have been in the past. Effectively, FPGAs allow a rapid soft reconfiguration of on chip hardware. Moreover, with a sufficient number of parallel operations, FPGAs can offer better performance-price and power dissipation than state of the art microprocessors or DSPs. Many implementations of lossy and lossless image compression encoders and decoders were performed on reconfigurable FPGA circuits [9], [10], [1]. In this paper, we discuss a hardware accelerated implementation of the PNG encoder for binary mask compression on FPGA. An optimized architecture is proposed as part of an hybrid software and hardware co-operating system. For its evaluation, the new designed PNG IP has been implemented on the *ALTERA* "*Arria II GX EP2AGX125EF35*" FPGA. The experimental results show a good match between the achieved compression ratio, the computational cost and the used hardware resources. The paper is organized as follows: first we present briefly the PNG encoder in section II. The performed PNG encoder optimizations, in order to well-ensure binary mask compression, are provided in section III. Section IV describe the proposed hardware design to accelerate the optimized PNG encoder on FPGA. Then, the obtained experimental results are discussed in section V. Finally, section VI concludes the discussion.

[1] $Demat+$ is a project established between the $SagemCom$ company and the $ESIEE\ Paris$ engineering school.

[2] JPEG [4] uses the discrete cosine transform, and JPEG2000 [6] uses the wavelet transform.

II. PNG ENCODER

The PNG format is very simple, where each PNG file is composed out of a *signature*[3] and *chunks*[4]. A valid PNG file consists at least of an *IHDR* chunk, one or more *IDAT* chunks, and an *IEND* chunk [5]. A minimal PNG file might look as shown in figure 1.

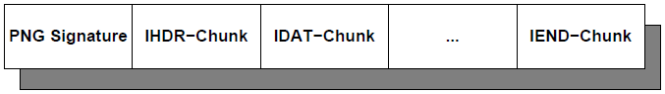

Fig. 1. Minimal PNG File

The PNG encoder [5] is principally composed of two main stages, namely a prediction step followed by a compression step. Indeed, PNG uses prediction in order to increase redundancy and therefor enhancing its compression rate. The prediction errors are then compressed using the *Deflate* algorithm [12]. Deflate is a lossless compression method which uses a combination of the LZ77 algorithm [14] and Huffman coding [11]. Deflate is independent of CPU type, operating system, file system and character set, and hence can be used for interchange [8]. The illustrated flowchart in figure 2 shows the involved steps in the PNG encoder.

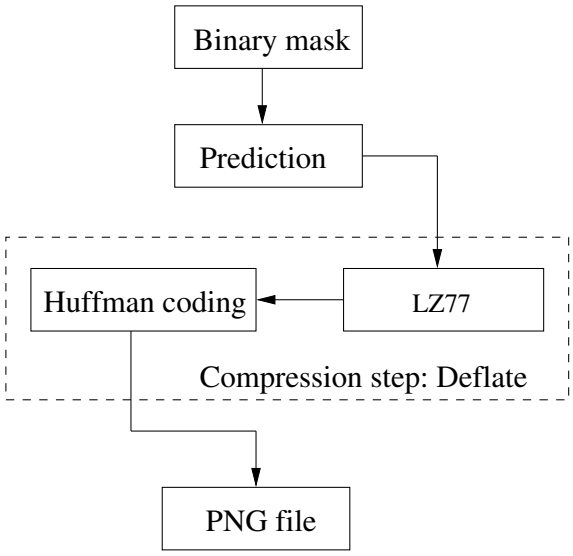

Fig. 2. Overview of the PNG encoder

In the following, we detail the different performed optimizations of the PNG encoder in order to well-ensure binary mask compression. To meet the standard defined by the format, the PNG file formatting can not be modified and remains unchanged regardless of the type of compressed data.

[3]The PNG file signature consists of eight bytes: <0x89 0x50 0x4e 0x47 0x0d 0x0a 0x1a 0x0a>.

[4]A chunk consist of four fields: <Length, Chunk Type, Chunk Data, and CRC>.

III. PROPOSED OPTIMIZATIONS OF PNG ENCODER FOR BINARY MASK COMPRESSION

A. Prediction stage

One of the key-features of PNG is the ability to chose out of five predictors (filters) for the predictive coding, namely: None (0), Sub (1), Up (2), Average (3) and Paeth (4). These filters are completely lossless and are applied to the raw image data before it is compressed and written to the output file. Every scan line of the image is preceded by a byte indicating the number of the used predictor among the five filter types.

PNG format do not impose any contrainte on the type of filter to use. In addition, filters do not have the same effectiveness on the used data. Therefore, the PNG prediction stage consists to determine which filter allows a better data compression. A suggestion in the PNG specification is to apply every filter to each scan line, compress it and check the size. This is not very fast but it is easy to implement and this approach results in the best possible compression ratio. On the other hand, the None filter is recommended for images of bit depth less than 8 [5]. For low-bit-depth grayscale images, particularly the binary masks in our case (only 1 bit depth), it may be a net win to transmit all scan lines unmodified; it is just necessary to insert the filter type byte "0" before the data.

In this context, we conducted a study to determine the best filter to use in this work. For this, we examined the impact of each filter on the compression rate of a panel of binary masks. Indeed, one filter may be more effective than all the others. In such case, we implement only the corresponding filter in the proposed hardware design in place of the whole prediction stage. Figure 3 illustrates the obtained sizes in *kilobytes* of the PNG coded binary masks. For all images, a single filter is used each time. For comparison reason, the standard PNG prediction is also used, and noted by *Pred.*

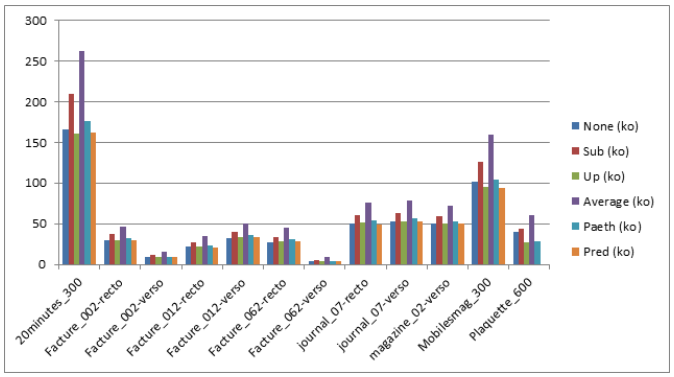

Fig. 3. Efficiency of various PNG prediction filters through a panel of binary masks

According to this study, we can deduce the efficiency of each filter depending on the obtained compression rates. Indeed, we can observe in figure 3 that both Sub and Average filters are less suitable to the treated masks. The Paeth filter and the standard PNG prediction are also excluded in our case. They have a high complexity degree and a great need for memory resources while they conducts to similar PNG compression results as the remaining filters. Finally, the choice between Up and None filters is to not apply any filtering because, based on the conducted study, these two filters have

almost the same compression results. This choice guarantees a competitive compression results while significantly simplifying our implementation. Hence, we maintain only the None filter in our hardware design, which amounts to not filter masks before *Deflation*.

B. Compression stage

After prediction, the coded pixel data is stored in a bit-stream which is subsequently deflated. As mentioned above, the Deflate algorithm [12] is a combination of LZ77 [14], [13] and Huffman coding [11]. It uses a variably sized sliding window and sorted hash tables to identify data patterns and compresses them using Huffman coding. We discuss in this section the optimizations made of the Deflate algorithm for binary mask compression.

1) LZ77 (Hash function): LZ77 [14] finds duplicated arbitrary sequence of bytes in the input data. The second occurrence of a string[5] is replaced by a pointer to the previous one, in the form of a pair <length, backward distance>, corresponding respectively to the length of the redundancy and the distance separating the two redundant strings. Distances are limited to 32 *kilobytes*, and lengths are limited to 258 *bytes*. When a string does not occur anywhere in the previous 32 *kilobytes*, it is emitted as a sequence of literal bytes. For this, linked lists are created to quickly find the redundant strings in dictionary and research is done by triplets of bytes to ensure a minimum compression rate. Depending on the available memory resources and the desired performance, the PNG compression can be carried out using various hash functions. For each string of three new bytes, a new hash value is computed based on the previous one and stored in hash table. As exposed by equation 1, the general formula of the possible hash functions remains the same, only the size of the final result and the shift made during the computation may change.

$$H = ((h << \textit{shift\_len}) \text{ XOR } C) \text{ AND } (2^{hash\_len} - 1) \quad (1)$$

With H : hash value of the new triplet.
h : hash value of the previous triplet.
C : last byte of the new triplet.
shift_len : shift bit number.
hash_len : hash bit number.

In the PNG specification [5], *shift_len* and *hash_len* are respectively equals to 6 and 15. Thus, computed hash values are coded on 15 bits (see table I), and the used hash table has a 32 *kilobytes* size. The standard used hash function, named in the following Hash_15, is given by equation 2:

$$H = ((h * 40_{16}) \text{ XOR } C) \text{ AND } 7FFF_{16} \quad (2)$$

With a view to optimize the needed memory resources and speed up the hash value computations, more adapted hash functions can be considered to treat binary masks. The greater the hash_len value is low, less is the required memory resources. However, fewer bits has the drawback of increasing the execution time. A statistical study to compute the triplets of bytes occurrence commonly encountered in binary masks was conducted. Based on the results of this study, two new hash functions Hash_10 and Hash_8 with respectively 10 and 8 bits hash_len are proposed. Hash_10 and Hash_8 formulas are shown respectively by equations 3 and 4:

$$H = ((h * 10_{16}) \text{ XOR } C) \text{ AND } 3FF_{16} \quad (3)$$

$$H = ((h * 8_{16}) \text{ XOR } C) \text{ AND } FF_{16} \quad (4)$$

Given that H, h and C in equation 2, 3 and 4 follow the same notation used in equation 1.

A summary of shift_len and hash_len values of the studied hash functions is provided in table I.

TABLE I. SHIFT_LEN AND HASH_LEN VALUES OF THE STUDIED HASH FUNCTIONS

hash function	*shift_len*	*hash_len*
Hash_8	3 ==> (2^3 = 8_{16})	8 ==> (2^8 - 1 = FF_{16})
Hash_10	4 ==> (2^4 = 10_{16})	10 ==> (2^{10} - 1 = $3FF_{16}$)
Hash_15	6 ==> (2^6 = 40_{16})	15 ==> (2^{15} - 1 = $7FFF_{16}$)

We note that, the obtained compression rates corresponding to Hash_8 and Hash_10 (equation 3 and 4) remain acceptable (see table III). However, a comparative study shows that Hash_10 provides a better compromise between runtime and compression rate. In addition, it still reducing significantly the needed memory resources (dictionary memory size becomes 1 *kilobyte* $< 2^{10} >$ instead of 32 *kilobytes* $< 2^{15} >$). Moreover, when coupled with optimal parameters[6], the selected hash function Hash_10 allows to further speed up the final hardware design.

2) Huffman coding: Huffman coding [11] is an entropy encoding algorithm used for lossless data compression. The principle of this coding is to use fewer bits for strings which appear more often in data stream and more bits for the less frequently ones. To avoid conflicts of code, Huffman coding respect the prefix property. That means there is no bit-sequence encoding which can be the prefix of any other bit-sequence. Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others of which find optimal prefix codes. In the PNG specification [5], the compressed data within the LZ77 data stream is stored as a series of blocks, each of which can represent LZ77-compressed data encoded with custom or fixed Huffman codes.

Custom Huffman code is computed dynamically based on the appearance frequency of the strings in the data stream. This kind of coding contributes, generally, to the improvement of the achieved compression rates. However, this is at the cost of system slowdown and implementation complexity. By against, as shown in table II, fixed Huffman code are obtained according to a static lookup table specific to the standard.

A comparative study of performance time, computational complexity and compression rate was conducted using respectively custom and fixed Huffman coding for PNG compression

[5] $String$ here must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.

[6] Parameters refer to the maximum number of searches to do "*MaxSearch*" and the length of the minimum redundancy to extract "*MinLength*".

TABLE II. STATIC HUFFMAN LOOKUP TABLE

Input byte values	*Number of bits*	*Fixed Huffman Codes*
0 - 143	8	[00110000 .. 10111111]
144 - 255	9	[110010000 .. 111111111]
256 - 279	7	[0000000 .. 0010111]
280 - 287	8	[11000000 .. 11000111]

of binary masks. To strengthen the choice of the selected hash function Hash_10, all three evaluated hash functions are employed in this study. Table III presents the obtained compression rates of this study. We note that the difference between the average compression rates, obtained with custom and fixed Huffman coding, is very insignificant. Indeed, against an average compression rate around "0.89" obtained with custom Huffman coding, fixed one leads to achieve "0.8689". Therefore, in order to speed up computing and reduce needed memory resources, we chose to implement the fixed Huffman coding.

TABLE III. PNG COMPRESSION RATES ACCORDING TO CUSTOM AND FIXED HUFFMAN CODING AND USING THREE DIFFERENT HASH FUNCTIONS

Images	**Compression rates**				
	Huffman	**Custom**	**Fixed**		
	Hash	**Hash_15**	**Hash_15**	**Hash_10**	**Hash_8**
20minutes_300		0,85	0,7992	0,7993	0,7993
facture_002-recto		0,88	0,8652	0,8650	0,8650
facture_002-verso		0,97	0,9552	0,9550	0,9552
facture_012-recto		0,92	0,8999	0,8999	0,8999
facture_012-verso		0,88	0,8467	0,8467	0,8467
facture_062-recto		0,89	0,8710	0,8707	0,8708
facture_062-verso		0,98	0,9744	0,9744	0,9744
journal_07-recto		0,82	0,7856	0,7857	0,7857
journal_07-verso		0,81	0,7777	0,7779	0,7779
magazine_02-verso		0,81	0,7893	0,7893	0,7893
mobilesmag_300		0,91	0,8789	0,8789	0,8789
plaquette		0,99	0,9839	0,9838	0,9838
Average		**0,89**	**0,8689**	**0,8689**	**0,8689**

Furthermore, for the used panel of binary masks and regardless of the used hash functions (Hash_15, Hash_10, and Hash_8), the obtained compression rate with the fixed Huffman coding are almost invariable. So, it is obvious that Hash_10, and Hash_8 have no effect on the compression rate, however they reduce the amount of used memory compared to the Hash_15 function. As mentioned in section III-B1, Hash_10 provides a better compromise between compression rate, runtime and used memory ressources, and therefore maintained in our design.

IV. ACCELERATED PNG ENCODER HARDWARE DESIGN

Based on the described optimizations in the previous section, the proposed hardware design in this work is a part of an hybrid software and hardware co-operating system.

As shown in figure 4[7], the PNG encoder design gathers a hardware implementation on FPGA and an external processor running a C program. These two elements have distinct roles:

- The Hardware implementation on FPGA performs data compression: optimized Deflate algorithm (no prediction, Hash_10, and fixed Huffman coding),

[7]The used color code in these figures is: Purple ==> Control Elements, Orange ==> Logical Elements, and Green ==> Memory Blocks.

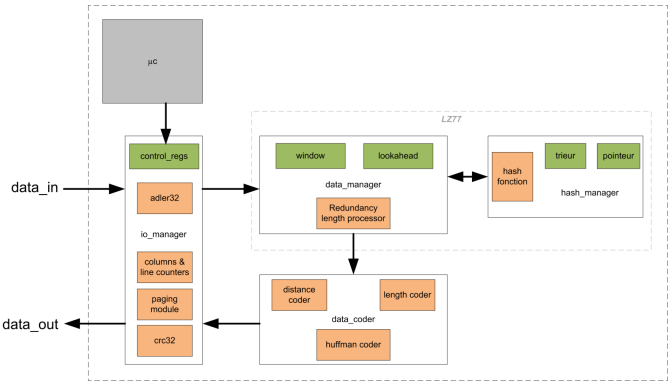

Fig. 4. Overview of the proposed PNG encoder design

- The external processor ensures the formatting of compressed data: issuance of PNG file header, the header of the IDAT chunks and the ancillary chunks. It launches also the PNG compression of masks.

In the following, we present the proposed hardware design to accelerate the PNG encoder on FPGA. Particularly the Input/Output, the Data, and the Hash managers, and the Data coder modules are described. The developed C code to ensure the software part is not described in this paper.

A. Input/Output manager

This module receives control signals from the processor (software part of the proposed design). Having the necessary informations[8] for the proper operation of the IP, it manages the inputs and outputs of the PNG encoder hardware part. Indeed, the Input/Output manager ensures an entry point of our design by collecting and filtering the incoming data, and an exit point by putting in shape the compressed data before transmission to the external system.

For each new incoming byte of uncompressed image, it updates four byte values (adler32) to be transmitted, sequentially, byte by byte to the LZ77 module. The Input/Output manager performs also the filtering of data, discussed at section III-A. We recall that in this work the used filter is the None one. Thus, byte "0" is just added in the beginning of each line. Finally, paging module is used to group compressed data by 32-bit words to send the output. Figure 5[7] shows an overview of the Input/Output manager module.

B. Data manager

The data manger module plays a central role within the proposed design. It allows to manage memories in which the redundancy research is made. For this, two memories are instantiated in the *data_manager_memories* block:

- window: contains the data dictionary on which the redundancy research is made,
- lookahead: contains the next bytes in the image that we look for matches in the window memory.

[8]Based on image dimensions received from the processor, lines and columns counters are used by the Input/Output manager module to determine if the entire image was read.

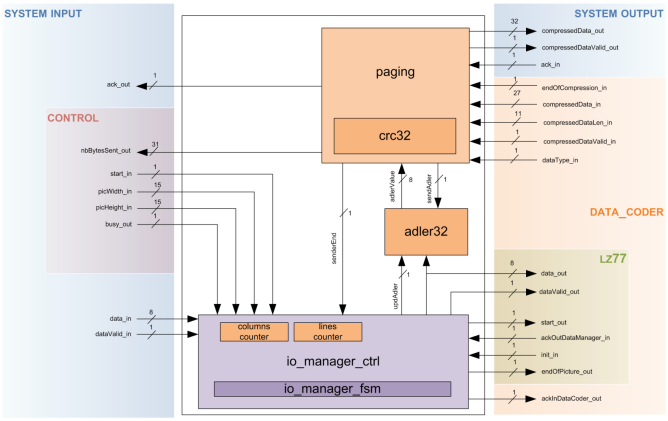

Fig. 5. Input/Output manager Overview

We use in our design a dictionary memory of 1024 bytes (due to the use of the Hash_10 function, see section III-B1) and a lookahead memory of 256 bytes. However, various configurations can be seen in the context of improving system performance. Figure 6[7] shows an overview of the data manager module.

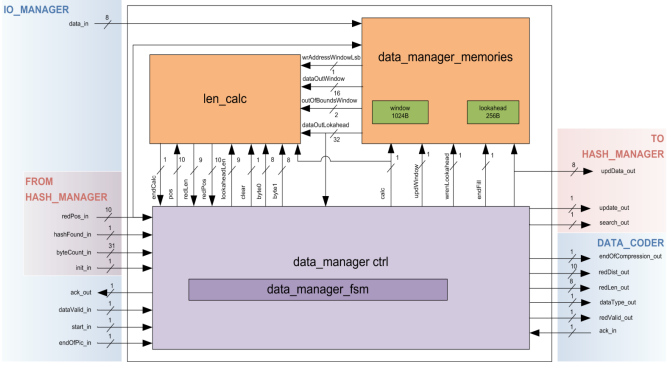

Fig. 6. Data manager Overview

The *data_manager_memories* block manages window and lookahead memories. Addresses of reading and writing are computed automatically based on the performed actions (update, redundancy research).

The *len_calc* block compares all received data from the *data_manager_memories* to determine the redundancy lengths. For the same triplet of bytes, only the maximum length of the obtained redundancies and the corresponding position thereto are stored.

The controller of the data manager module cares about its proper operation but also acts as a master for the hash manager module (see following subsection IV-C). The main actions of the *data_manager ctrl* block are:

- Obtain research positions from hash manager,
- Start and stop redundancy researches,
- Update window and lookahead Memories,
- Order the update of the hash manager module.

C. Hash manager

The hash manager module manages, essentially, the two used hash tables *trieur* and *pointeur*:

- Trieur: this table is addressed according to the hash values of the triplets to index. It stores the position of the last occurrences of each hash value. These positions point elements of the *pointeur* hash table,
- Pointeur: each cell of this table contains the position of the next occurrence of the hash value originally pointed to by the *trieur* hash table. Stored values in this table are therefore pointers to other cells in the same table.

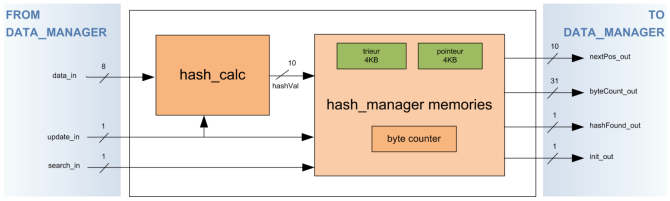

Fig. 7. Hash manager Overview

Through the received order from data manager, hash manager module can update hash tables or perform researches thereto. Figure 7[7] shows an overview of the data manager module.

The *hach_calc* block receives the triplet of bytes to encode byte by byte throw the *data_in* input signal. It ensures the computation of the corresponding hash value according to the employed Hash_10 function.

D. Data coder

It may be difficult to achieve a hardware coding according to the provided tables in the deflate method [12]. using lookup tables.

To obtain the codes corresponding to lengths and distances so this module uses a sequential method to determine the range of values in which are the distance and length. The algorithms used are described below.

Figure 8[7] shows an overview of the data manager module.

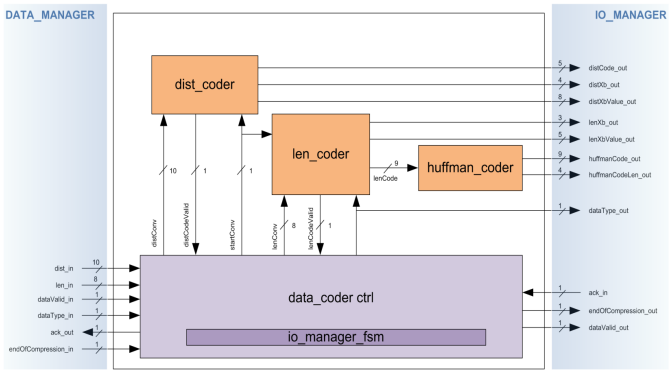

Fig. 8. Data coder Overview

V. Results

For evaluation, the proposed PNG IP has been implemented on the FPGA board *ALTERA*: "Arria II GX EP2AGX125EF35". Only simple formatting tasks were deported on the *NIOS* processor. In order to optimize the data path of the LZ77 module and obtain an optimal compression speed, two parameters were identified and assessed: *MaxSearch* and *MinLength*. They respectively indicate maximum number of searches to do and the length of the minimum redundancy to extract.

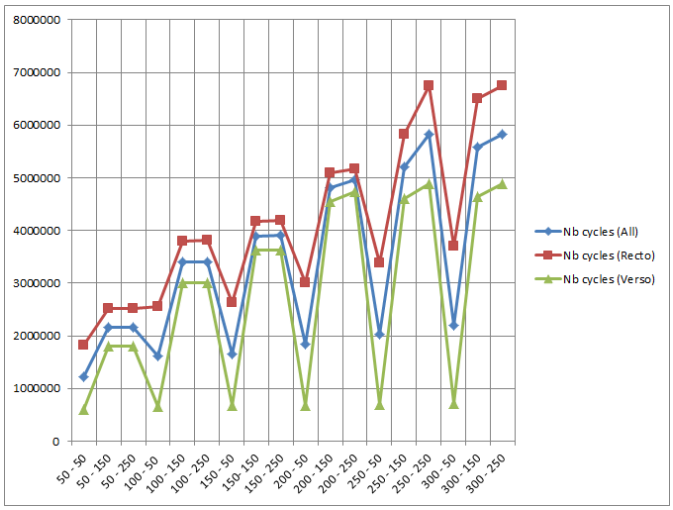

Fig. 9. Average Clock Cycles of the proposed PNG encoder architecture according to combinations of $MaxSearch$ and $MinLength$ parameters

Un panel de factures recto-verso a t utilis pour les tests. Les masques binaires de ces factures sont extraits, en 150 ppp, l'aide de la mthode de segmentation en couches fond/forme $DjVu$ [7]. Les rsultats de compression PNG de ces masques, obtenus avec l'architecture propose, sont illustrs par les figures 9 et 10. Ces deux figures affichent respectivement les nombres de cycles moyens et les taux de compression moyens de l'architecture PNG propose en fonction d'un ensemble de combinaisons des paramtres $MaxSearch$ et $MinLength$. Les exprimentations sont ralises en utilisant six valeurs de $Max-Search$: 50, 100, 150, 200, 250 et 300 et trois valeurs de $MinLength$: 50, 150 et 250.

Les courbes illustres par les figures 9 et 10 reprsentent des factures verso (triangles verts), recto (carrs rouges) et recto-verso, indiqu par *All* dans la lgende des courbes, (losanges bleus). Souvent, les faces recto des factures contiennent plus de donnes que celles du verso. Ceci explique, comme le montre les figures 9 et 10, l'augmentation du nombre de cycles de compression PNG dans le cas des faces recto et leur faible taux de compression par rapport aux faces verso.

Les rsultats obtenus montrent que $MinLength = 50$ pour toutes valeurs de $MaxSearch$ assure de meilleurs temps d'excution c'est--dire moins de cycles de calcul (voir figure 9). Toutefois, des valeurs plus faibles de $MaxSearch$ acclre le processus de compression. En effet, la rduction de la longueur minimale permet de trouver des redondances plus rapidement. De plus, limiter le nombre de recherches maximal permet d'interrompre les recherches assez longues sans avoir pour autant un grand impact sur les taux de compression dans

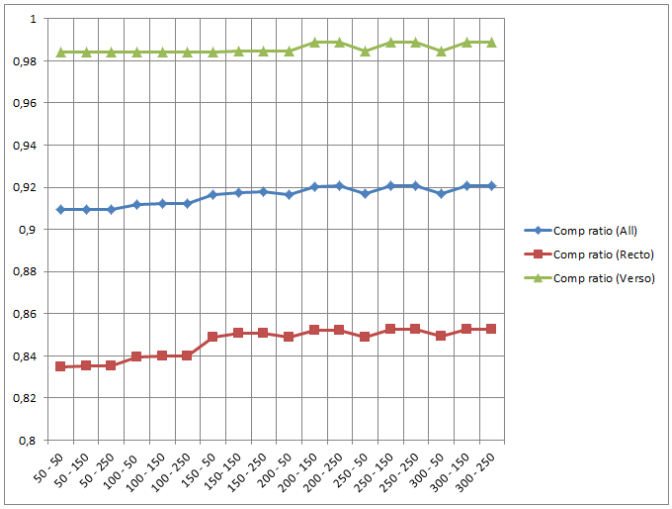

Fig. 10. Average compression rate of the proposed PNG encoder architecture according to combinations of $MaxSearch$ and $MinLength$ parameters

le cas des images binaires testes (voir figure 10). Ainsi, la combinaison optimale choisie lors de nos exprimentations est :
$<MaxSearch = 50, MinLength = 50>$.

VI. Conclusion

This paper presented an optimized design of PNG encoder, with the aim of accelerating the binary mask compression on FPGA. Firstly, the PNG encoder was studied and obtained compression rates on binary masks were assessed. According to this study, a set of optimization was introduced namely the non use of the prediction stage, the proposition of a new hash function and the choice of the more suitable Huffman coding. As a part of an hybrid co-operating system, we proposed a new hardware design of the optimized PNG encoder. The experimental results was conducted on the "Arria II GX EP2AGX125EF35" *ALTERA* FPGA board.

The experimental results show

A good match between achieved compression rate and consumption in terms of computational costs, memory area and number of used logic elements.

Acknowledgment

The authors would like to thank Mr. Jérôme Berger, image and signal processing engineer at $SagemCom$ company and Mr. Geoffroy Marpeau engineer at $ESIEE\ Paris$, they are very kind and patient.

References

[1] Huang, Shizhen, and Tianyi Zheng. *Hardware design for accelerating PNG decode*. Electron Devices and Solid-State Circuits, 2008. EDSSC 2008. IEEE International Conference on. IEEE, 2008.

[2] Michel Chilowicz. *Une synthse sur les formats usuels d'images numriques fixes*. Rapport de recherche 2006.

[3] Graphics Interchange Format (GIF) specification. *URL http://www.fileformat.info/format/gif/spec/index.htm.*

[4] International Telecommunication Union (ITU). *URL http://www.w3.org/Graphics/JPEG/itu-t81.pdf.*

[5] G. Randers-Pehrson, et. al. *PNG (Portable Network Graphics) Specification Version 1.2*. URL http://www.libpng.org/pub/png/ PNG Development Group, July 1999.

[6] Cs Msu Graphics&Media Lab Video Group. *JPEG 2000 Image Codecs Comparison*. Moscow, September 2005.

[7] Lon Bottou, Patrick Haffner, Paul G. Howard, Patrice Simard, Yoshua Bengio, and Yann Lecun. s.l. *High quality document image compression with djvu*. Journal of Electronic Imaging, 1998, Vol. 7, pp. 410-425.

[8] RFC-1951, *Deflate Specification*.

[9] Sanjeevannanavar, S., Nagamani, A.N. *Efficient design and FPGA implementation of JPEG encoder using verilog HDL*. Nanoscience, Engineering and Technology (ICONSET), 2011 International Conference on, vol., no., pp.584-588, 28-30 Nov. 2011

[10] Daryanavard, H., Abbasi, O., Talebi, R. *FPGA implementation of JPEG-LS compression algorithm for real time applications*. Electrical Engineering (ICEE), 2011 19th Iranian Conference on , vol., no., pp.1-4, 17-19 May 2011

[11] Huffman, D. *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the IRE 40 (9): 10981101, 1952. doi:10.1109/JRPROC.1952.273898

[12] Peter, Deutsch L. Deflate *Compressed Data Format Specification version 1.3. gzip.org*, 1996. URL: http://www.gzip.org/zlib/rfc-deflate.html.

[13] P. Deutch, J.-L. Gailly, and M. Adler. *GZip URL http://www.gzip.org*.

[14] Ziv J., Lempel A., *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, 1977, Vol. 23, No. 3, pp. 337-343.